Informatika 3
3. előadás

Tóth Dávid

Budapesti Műszaki és Gazdaságtudományi Egyetem

2026.03.03.

Osztályok

- A C++-ban van lehetőség új adattípusok létrehozására.
- Erre az ún. osztályokat (`class`) használjuk.
- Az osztály leírásában megadjuk az osztály által tárolt adatot, továbbá azokat a tagfüggvényeket (metódusokat), amikkel az objektumot vizsgálhatjuk és amelyekkel változtatásokat hajthatunk rajta végre.
- Mindezt egy `Point` adattípus példáján fogjuk végigkövetni, amely a sík egy pontját reprezentálja.

Osztályok

A `Point` adattípuson a következő műveleteket szeretnénk elvégezni:

- lekérdezni a pont Descartes-koordinátáit,
- lekérdezni a pont polárkoordinátáit,
- megváltoztatni a pont egyik vagy mindkettő Descartes-koordinátáját,
- megváltoztatni a pont egyik vagy mindkettő polárkoordinátáját,
- elforgatni a pontot az origó körül egy megadott szöggel,
- ellenőrizni, hogy két pont megegyezik-e,
- két pont közötti távolságot meghatározni,
- két pont közötti felezőpontot meghatározni,
- kiíratni a `P` pont koordinátáit a `cout` $\ll$ `P` paranccsal.

Osztályok definiálása

- Egy osztály definíciója a következőképp néz ki:

```
class OsztalyNev {
    adatok es tagfuggvenyek deklaracioja;
};
```

- Az osztályok neveit nagy betűvel szokás kezdeni (nem kötelező, de ajánlott).
- Ne felejtsük el a `;`-t a záró kapcsos zárójel után!
- A definíciót egy külön fejállományba írjuk (pl. a `Point` osztály esetén a `point.h`-t használjuk).

Adatok elrejtése

- Az osztály által reprezentált objektumot leíró adatot ún. *tagváltozókban* tároljuk.
- Erre a célra már létező típusokat használunk.
- Pl. a `Point` osztályban a pont koordinátáit `double` típusú változókban tároljuk:

```
class Point {
    double x;
    double y;
};
```

- A fenti deklarációval az x és y változóknak nem foglalódik memória, ez csak akkor történik meg, ha majd létrehozzuk ezen osztálynak egy (vagy több) *példányát*:

```
Point X;
Point Y;
```

Adatok elrejtése

- Az osztály minden egyes példányához külön `x` és `y` tagváltozók fognak tartozni: `X.x` jelöli az `X` példány `x` koordinátáját, `Y.x` pedig az `Y`-ét.
- A tagváltozókra `peldanyNev.tagvaltozoNev` szintaxissal hivatkozunk.
- Általában nem jó, ha a felhasználó közvetlenül hozzáfér az osztályt leíró adatokhoz, ezért ezeket tipikusan `private` kulcsszó után írjuk:

```
class Point {
private:
    double x;
    double y;
};
```

- Az így deklarált adatok nem érhetők el és nem módosíthatók közvetlenül az osztályon kívülről.

Adatok elrejtése

- Az adatok elérése publikus tagfüggvényeken keresztül történhet, ezeket `public` kulcsszó után írjuk:

```
class Point {
private:
    double x;
    double y;
public:
    double getX()const;
};
```

- A fenti példában a `getX()` tagfüggvény feladata, hogy az pont `x` koordinátáját visszaadja. (A fenti függvénydeklarációról később.)
- A `public` kulcsszó után írt változók és tagfüggvények elérhetők az osztályon kívülről (pl. `X.getX()`).

Adatok elrejtése

- Ez a megoldás biztosítja azt is, hogy ha az adatok reprezentálása később (pl. hatékonysági okokból) megváltozik, akkor is érvényes maradhat ugyanaz a kód.
- Pl. ha a pontot polárkoordinátákkal szeretnénk leírni, akkor az `x` és `y` változók helyett pl. `r` és `A` változókat deklarálhatunk, de ettől a `getX()` tagfüggvény módosított működéssel, de továbbra is az `x` Descartes-koordinátát adhatja vissza, így a korábban írt kód továbbra is érvényes marad.
- A publikus tagfüggvényeken keresztül vizsgáljuk és változtatjuk az objektumot.
- Nem kell tudnunk, hogy a tagfüggvény hogyan működik, csak hogy mit csinál.
- Egy másik ok az adatok elrejtésére: a felhasználó esetleg olyan módon változtathatja a tagváltozókat, hogy azok az objektum egy érvénytelen állapotát írják le.

Konstruktorok

- A *konstruktor* egy olyan függvény, ami az osztály egy példányának létrehozásánál meghívódik.
- Pl. az

  `OsztalyNev X;`

  kód az `OsztalyNev()` konstruktort hívja meg.
- A konstruktorok neve meg kell egyezzen az osztály nevével.
- A konstruktoroknak nincs visszatérési értéke (így visszatérési típusuk sincs).

Konstruktorok

- Egy osztálynak lehet több konstruktora is, ezeknek az argumentumokban különböznie kell.
- Konstruktor nélkül is megírhatunk egy osztályt, ekkor egy üres default konstruktor fut le egy példány létrehozásánál.
- Egy konstruktor lehet publikus, de szükség esetén private is.
- Pl. a `Point` osztályban a következő publikus konstruktorokat deklaráljuk:

```
Point();
Point(double xx, double yy);
```

- A második konstruktort arra fogjuk használni, hogy rögtön a deklarálásnál inicializáljuk a változót.

Definiálás

- A konstruktorokat és később a tagfüggvényeket egy külön `.cpp` file-ban definiáljuk.
- Ennek neve megegyezik a fejállomány nevével. Pl. `point.h` esetén ez `point.cpp`, és ennek tartalmaznia kell az

  `#include "point.h"`

  sort, hogy a fordító tudjon a `Point` osztály definíciójáról.
- A `Point()` konstruktor definíciója:

```
Point::Point() {
    x = .0;
    y = .0;
}
```

- Itt az első `Point` arra utal, hogy a `Point` osztály egy konstruktorát/tagfüggvényét definiáljuk, és `::` után írjuk a konstruktor nevét és argumentumait, majd a definíciót.

Definiálás

- A `Point` másik konstruktorának definíciója:

```
Point::Point(double xx, double yy) {
    x = xx;
    y = yy;
}
```

- Itt az `x`, `y` mindig az adott példány `x`, `y` változójára vonatkozik.
- Bár a C++ engedi, soha ne nevezzük a paramétereket `x`-nek és `y`-nak, mert akkor ezek elfedik az objektum adattagjait.
- Megszokott a `_x` vagy `y_` megoldás is az elnevezésnél.
- Ez a konstruktor a deklarációnál így hívható meg:

```
Point P(0.5,0.3);
```

Definiálás

- Ha mégis ugyanúgy nevezzük a konstruktor/tagfüggvény paramétereit, mint az adattagokat, akkor az adattagok a `this` pointer segítségével érhetők el, amely az osztály adott példányának memóriacímére mutat.
- Ha a `this` pointert használjuk, akkor a tagváltozók/tagfüggvények `"."` helyett a `"->"` karaktersorozat segítségével érhetők el.
- Ha a `*this` segítségével magára az adott példányra hivatkozunk, akkor használható a `"."` karakter.

Definiálás

- Az `x` ill. `y` nevű adattagokkal rendelkező `Point` osztály következő konstruktorának működése helyes:

```
Point::Point(double x, double y) {
    this->x = x;
    this->y = y;
}
```

- Ugyanez így is írható:

```
Point::Point(double x, double y) {
    (*this).x = x;
    (*this).y = y;
}
```

Inicializáló lista

- Egy osztály egy konstruktora először az objektum tagváltozóit inicializálja, mielőtt még a konstruktor törzse lefutna.
- Beépített típusoknál (`double`, `bool`, `int` stb.) ez semmit nem csinál, de osztály típusoknál meghívódik a default konstruktor.
- Ha ezt nem tudja automatikusan megtenni, vagy pedig felül akarjuk bírálni az alapértékeket, akkor az inicializáló listát használjuk, ennek szintaxisa:

```
class C{
    típus a, b;
    C() : a(paraméterek), b(paraméterek) {}
};
```

Inicializáló lista

- Az inicializáló lista előbb fut le, mint a konstruktor törzse.
- Azokat az adattagokat, amiket nem adunk meg az inicializáló listában, a fordító alapértelmezett konstruktorral inicializálja (vagy beépített típus esetén sehogy).
- Ha pedig ez nem lehetséges, akkor hibát kapunk.
- Az inicializáló listát *kell* használnunk az alábbi esetekben:
  - referencia adattagot kéne inicializálni (nem tudja a fordító, hogy mire hivatkozzon a referencia)
  - konstans adattagot kéne inicializálni (nem tudja a fordító, hogy milyen értékre akarjuk beállítani)
  - objektum adattagot kéne inicializálni (meghívódik a konstruktora), de nincs publikus default konstruktora (nem tudja a fordító, hogy milyen értékekkel hívja a konstruktorát) vagy attól eltérő paraméterezéssel szeretnénk meghívni

Inicializáló lista

- Az inicializáló listában fontos az adattagok sorrendje, ez mindig legyen a struktúra definíciója szerinti sorrend.
- Valójában bármilyen sorrendben odaírhatjuk az adattagokat, de mindig a struktúra definíciója szerinti sorrendben történik az inicializálás.
- Ha a leírt sorrendtől eltérő sorrendben történik valami, könnyen írhatunk hibás kódot, ezért mindig tartsuk a megfelelő sorrendet!
- Kerülendő, de inicializáló listánál működik az a megoldás, ha ugyanaz a paraméter neve, mint a tagváltozóé (hiszen itt világos, hogy mi jelöl tagváltozót, és mi jelöl paramétert):

```
Point(double x, double y) : x(x), y(y) {}
```

Értékadás és konverzió

- Tekintsük a következő kódot:

```
Point P;
Point Q(-2., 4.);
P = Q;
```

- Ha az = operátort máshogy nem definiáljuk (lásd később), akkor a C++ a `Q` változó tagváltozóiban lévő értékeket sorban átmásolja a `P` változó tagváltozóiba (akkor is ha private változókról van szó), tehát a 3. sor után `P.x` értéke -2, és `P.y` értéke 4.

Értékadás és konverzió

- Ha konverziót szeretnénk végrehajtani, azt nekünk kell egy alkalmas konstruktorral megvalósítani.
- Ha pl. egy `double x` változó értékét `Point`-tá szeretnénk konvertálni úgy, hogy `x`-ből az `(x,.0)` pont legyen, akkor hozzunk létre egy `Point(double xx)` konstruktort, ami az `x` tagváltozó értékét `xx` értékére állítja, az `y` tagváltozó értékét pedig 0-ra.
- Ekkor egy konverzió:

```
Point P;
double x;
...
P = Point(x);
```

- A fenti kód utolsó sora létrehoz egy Point objektumot a megfelelő konstruktorral, és annak értékét átmásolja `P`-be.

Tagfüggvények

- Tagfüggvényeken keresztül érhetjük el az objektumhoz tartozó adatokat, és ezekkel végezhetünk különféle műveleteket az objektumokkal.
- A tagfüggvényeknek lehet visszatérési értékük, de deklarálhatjuk `void`-nak őket, ekkor nem adnak vissza értéket.

Tagfüggvények

- Ha a tagfüggvények az objektumot nem változtatják meg, akkor `const`-nak deklaráljuk őket. Ez nem kötelező a fordító miatt, de mindig így járjunk el!
- Ez a gyakorlat számos hibát már fordítási időben felfed.
- Ha egy függvényt konstansnak deklarálunk, akkor annak definíciója valóban nem változtathatja meg a tagváltozó értékét, a fordító jelezni fog egy ilyen kódnál.
- Konstans tagfüggvény a tagváltozóknak vagy az adott objektumnak csak konstans tagfüggvényét hívhatja, konstans objektumoknak csak konstans tagfüggvényei hívhatók, ezért is érdemes mindent konstansnak deklarálni, amit lehet.

Tagfüggvények

- Pl. a `Point` osztály `getX()` függvényét konstansnak deklaráljuk, mert ez nem változtatja az `x` koordinátát, csak visszaadja azt.

```
class Point {
...
public:
    double getX()const;
...
};
```

- A definíciója

```
double Point::getX()const {
    return x;
}
```

Tagfüggvények

- Pl. a `Point` osztály `setX(double xx)` tagfüggvénye beállítja az `x` koordinátát az `xx` értékére, ezért ez nem lehet konstans.

```
class Point {
...
public:
    void setX(double xx);
...
};
```

- A definíciója

```
double Point::setX(double xx) {
    x = xx;
}
```

Tagfüggvények

- Természetesen deklarálhatunk függvényeket az osztályon kívül is, amik az adott osztályt használják, ezeknek a definíciójában persze nem kell az `OsztalyNev::`
- Pl. a `midpoint` függvény kiszámolja a két pontot összekötő szakasz felezőpontját.
- Deklaráció:

```
Point midpoint(Point P, Point Q);
```

- Definíció:

```
Point midpoint(Point P, Point Q){
    ...
}
```

Operátorok

- A C++-ban számos operátor definíciója kiterjeszthető, ezt *operátor túlterhelésnek* nevezzük.
- A `Point` osztályban az ==, `!`= operátorokat terheljük túl a objektumok egyenlőségének ellenőrzéséhez, és a ≪ operátort a kiíráshoz.

Operátorok

- Az == operátornak két argumentuma van, egy a bal, egy pedig a jobb oldalán, a visszatérési értéke pedig `bool`.
- A deklarációt az osztályon belül végezhetjük el:

```
class Point {
    bool operator==(const Point& Q)const;
};
```

- Az operátor visszatérési értéke `bool`, neve `operator==`.
- Egy paraméter van, ez áll az egyenlőség jobb oldalán, míg a bal oldalon lévő objektum tagfüggvénye hívódik meg.
- A paraméter `const Point& Q`, ez azt jelzi, hogy egy `Point` típusú változóra hivatkozó referenciát kapunk.
- A `const` kulcsszó a parméter előtt jelzi, hogy a paramétert nem változtatjuk, a második `const` szerint pedig az objektum, aminek a tagfüggvénye meghívódik, nem változik.

Operátorok

- Megjegyzés: az == operátort deklarálhattuk volna az osztályon kívül, ekkor két paramétere lenne.
- A != operátor analóg módon deklarálható/definiálható.
- Szokásos trükk: ha az == operátor már definiálva van, akkor a != operátor definíciója lehet

```
bool Point::operator!=(const Point& Q)const {
    return ! ((*this) == Q);
}
```

- Itt `this` az adott objektumra (a példányra) mutató pointer, így `*this` magára az adott példányra vonatkozik, aminek a tagfüggvényét meghívjuk. Ezt bármely tagfüggvényben használhatjuk.

Operátorok

- Ki szeretnénk írni a pont koordinátáit a `cout << P` paranccsal.
- Az `operator<<` nem deklarálható az osztályon belül, mert a bal oldalán a `cout` objektum áll, aminek a típusa nem `Point`.
- A `cout` típusa `ostream`, ez az `iostream` fejállományban van deklarálva.
- Ahhoz, hogy pl. a

  ```
  cout << P << endl;
  ```

  kód működjön, az első művelet egy `ostream`-re mutató referenciát ad vissza, amire aztán a második << operátor alkalmazható.
- Tehát a deklaráció

  ```
  ostream& operator<<(ostream& os, const Point& P);
  ```

Operátorok

- Visszatérési értékként pedig ugyanazt az `ostream` objektumot adjuk vissza, amit megkaptunk paraméterként, így ugyanabba folytatólagosan írhatunk.
- A definíció:

```
ostream& operator<<(ostream& os, const Point& P) {
    os << "(" << P.getX() << "," << P.getY() << ")";
    return os;
}
```

Operátorok

- A +/- operátorokat is túlterhelhetjük, ha a pontokat mint vektorokat szeretnénk összeadni/kivonni:

```
Point Point::operator+(const Point& Q) const{
    return Point(x + Q.x, y + Q.y);
}
```

- A skalárral való szorzást az osztályon kívül kell definiálnunk, ha balról szeretnénk szorozni a skalárral (ha az osztály tagfüggvényeként definiálunk egy operátort, akkor a bal oldalon mindig az a példány áll, aminek a tagfüggvényét meghívjuk).

Operátorok

- Az =, +=, -=, *=, /= stb. operátorokat ugyanúgy túlterhelhetjük, mint a többit, a visszatérési érték ezekben az esetekben mindig legyen referencia.
- Egyrészt így elkerülhető a változók értékének felesleges másolása, másrészt az eltérő gyakorlat könnyen hibákhoz vezethet. Pl. az alábbi kód nem fordul le:

```
class C{
    ...
    C operator=(const C& other){...}
    C& operator+=(const C& other) {
        return (*this) = (*this) + other;
    }
};
```

- Az = operátor egy `C` típusú értékkel tér vissza, ez a += függvény futása után nem tárolódik tovább a memóriában, így referenciaként nem adható vissza.

Operátorok

Alapelv: ha új objektum keletkezik, térjünk vissza érték szerint, ha nem, akkor referencia szerint.